Concurrency, Intuition and Formal Verification: Yes, We Can!

A Position Paper: *"Curricula for Concurrency and Parallelism"* (SPLASH 2010)

Jan B. Pedersen, School of Computer Science, UNLV, USA (*`matt@cs.unlv.edu`*)
Peter H. Welch, School of Computing, University of Kent, UK (*`phw@kent.ac.uk`*)

Abstract

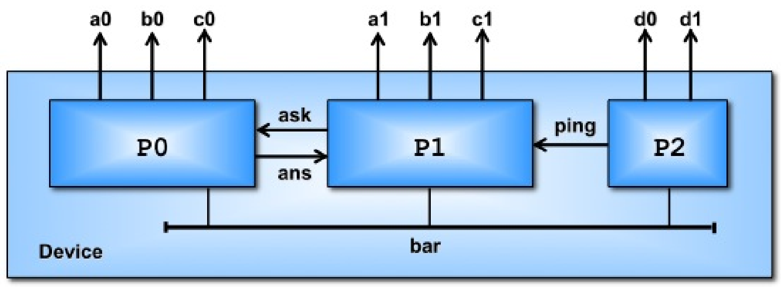
Not only can (and should) concurrency be introduced early in the undergraduate CS curriculum – but mechanisms for its formal analysis and verification can also be presented that are intuitive, effective and easy to learn and apply. Further, this can be done without requiring students to be trained in the underlying formal mathematics. Instead, we stand on the shoulders of giants who have engineered the necessary mathematics into the concurrency models we use (CSP, π-calculus), the programming languages/libraries (occam-π, JCSP) that let us design and build efficient executable systems within these models, and the model checker (FDR2) that lets us explore and verify those systems. All we require from our students are a love of the subject, a flair for programming and some time and effort. This position paper presents some experience over the past year that lets us make these claims.

Motivation

Multi-core architectures are now standard, with the number of cores per processor growing each year. Multi-processor networks are inescapable for super-computing problems and many (most?) forms of embedded computer platform. Programmers (and students) cannot avoid concurrent reasoning when dealing with these devices – avoidance leads to many bad things. Verification of this concurrent reasoning is mostly set aside (as it has generally been for sequential reasoning, we admit). A significant amount of professional development time and money is spent instead on *testing* software. However, testing and debugging concurrent programs is even more difficult than sequential programs – common faults are intermittent and not reproducible on demand. If the concurrency pattern is beyond the *embarrassingly parallel* (i.e., the processes need to engage with each other) and we have made some mistakes in design or coding, testing may never see these faults … and our system will eventually fail in service. So, we need to verify. Now, just as we need tools (e.g. programming languages) to produce *executable* systems, we need tools (e.g. model checkers) to produce *verified* systems. Language and model checker pairs need to live to the same concurrency model.

Eaxmple

The diagram shows the internal structure of **`Device`**, a component of a robot control system driving 8 output channels. There are 3 sub-components (**`P0`**, **`P1`** and **`P2`**) running concurrently. They exchange information using internal channels (**`ask`**, **`ans`**, **`ping`**) and all coordinate their actions on an internal barrier (**`bar`**).



CSP [1, 2] semantics apply. Channel communication is unbuffered: the sender process must wait for the receiver – and vice-versa. Barrier synchronisation means that any process engaging on the barrier must wait until all processes (plugged into the barrier) engage – the last one unblocks them all.

We show two representations defining the behaviour of `Device`: one in occam-π [3] (for compiling to a runnable system) and one in CSP (for formal analysis). The representations are in 1-1 correspondence and our students have had little trouble shifting between them. A tool, [4], exists to generate the CSP automatically from occam-π, but this is not yet ready for use in the classroom.

Behaviour questions we wish to check are: might the `Device` *deadlock* (stop); might it *livelock* (engage in an infinite sequence of internal channel or barrier synchronisations between its sub-processes, with no further outputs signaled); is it *safe* (producing no incorrect sequences of signals); will it stay *alive* (offering all permitted signal sequences)?

For our example, data values and computations are not relevant. For simplicity, they are omitted in the following, with all message content abstracted to zero. Here is the executable (occam-π) code:

```
PROC P0 (CHAN INT a0!, b0!, c0!,
         CHAN INT ask?, ans!, BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      ask ? x      -- take question
      a0 ! 0
      ans ! x      -- return answer
      b0 ! 0
      SYNC bar     -- wait for the others
      c0 ! 0
:
```

```
PROC P1 (CHAN INT a1!, b1!, c1!,
         CHAN INT ask!, ans?, ping?, BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      ask ! 0      -- ask question
      ans ? x      -- wait for answer
      a1 ! 0
      b1 ! 0
      SYNC bar     -- wait for the others
      c1 ! 0
      ping ? x     -- wait for more info
:
```

```
PROC P2 (CHAN INT d0!, d1!, ping!, BARRIER bar)
  WHILE TRUE
    SEQ
      SYNC bar     -- wait for the others
      d0 ! 0
      ping ! 0     -- update neighbour
      SYNC bar     -- wait for the others
      d1 ! 0
      ping ! 0     -- another update
:
```

```
PROC Device (CHAN INT a0!, b0!, c0!,
             CHAN INT a1!, b1!, c1!, d0!, d1!)
  CHAN INT ask, ans, ping:
  BARRIER bar:
  PAR ENROLL bar
    P0 (a0!, b0!, c0!, ask?, ans!, bar)
    P1 (a1!, b1!, c1!, ask!, ans?, ping?, bar)
    P2 (d0!, d1!, ping!, bar)
:
```

What patterns of signal are possible from `Device`? *Intuition:* first is `a0` (only external channels matter). This comes from `P0`, which has just been asked a question by `P1`. `P1` can't signal on `a1` until it has an answer from `P0`. `P2` can't get past its first barrier, `bar`. What line signals second? Either `b0` or `a1`. If it was `b0`, then `a1` will definitely be third and `b1` fourth. If second was `a1`, then third is either `b0` or `b1` and fourth is whichever was not third. Then, the internal barrier (`bar`) happens, as all three processes reach it, and the fifth, sixth and seventh are `c0`, `c1` and `d0` in any order. That's 18 possible orderings of the first 7 signals. But what happens when the sub-processes start looping? Could `P0` signal again on `a0` *before* `P2` gave its first `d0`? Are there any more first-7 signal sequences?

We can *formally verify* the above intuition, and answer the open questions, with a CSP representation of the system. We write in CSP-M, the machine readable form used by FDR2 [5]. CSP treats channel communications and barriers in the same way: they are all *events* (declared as `channel`s in CSP-M). For our example system, we can abstract the channel communications further by omitting the data sent (always zero) and the direction of communication (which is irrelevant to this formal analysis):

```
channel a0, b0, c0, a1, b1, c1, d0, d1, ask, ans, ping, bar

P0 = ask -> a0 -> ans -> b0 -> bar -> c0 -> P0
P1 = ask -> ans -> a1 -> b1 -> bar -> c1 -> ping -> P1
P2 = bar -> d0 -> ping -> bar -> d1 -> ping -> P2

P0P1 = (P0 [| {ask, ans, bar} |] P1) \ {ask, ans}
Device = (P0P1 [| {ping, bar} |] P2) \ {ping, bar}
```

Note that CSP-M is a *declarative* (or *functional*) language, whereas occam-π is *imperative*. Students who love to program have no problem learning new syntax and semantics, so long as they understand why *(occam-π is for building executables; CSP-M is for reasoning about them; they have the same concurrency semantics)*.

In the above, the process loops in occam-π become tail recursion in CSP-M. The *parallel* operator in CSP-M, `[|` ***sync-set*** `|]`, is *binary* – hence, the `Device` network is built in stages, two processes at a time. For an event in the ***sync-set*** (of the parallel operator) to occur, both process operands must engage. The hiding expression, ***process \ hide-set***, makes events in the ***hide-set*** locally declared within ***process*** (i.e., unrelated to same-named events elsewhere).

With the CSP definition of **Device**, we can start asking questions. Loading it into the FDR2 *GUI*, we straight away discover it is free from deadlock and livelock (which CSP calls *divergence*), simply by clicking the buttons labeled to perform these checks. To check whether particular event sequences may initially be performed by **Device**, define processes that have no choice in the matter – e.g.

```
T0 = a0 -> b0 -> a1 -> b1 -> d0 -> c0 -> c1 -> STOP
T1 = a0 -> b0 -> a1 -> d0 -> b1 -> c0 -> c1 -> STOP
```

Now, ask whether each of these *trace refines* **Device**. FDR2 reports that **T0** does, which means that any trace (sequence of events) it performs can also be performed by **Device**. FDR2 reports that **T1** does not, which means that one (at least) of its traces cannot be performed by **Device**. Comparing the two test processes, we can immediately deduce the fault lies in the mis-ordering of **b1** and **d0**.

Let's ask a more difficult question about the continuously running system. Suppose the robot would do something *very bad* if its controller **Device** were ever to signal twice on **a0** without a signal on *either* **d0** *or* **d1** in between – an in-service failure. Might this happen? Simple: write a process that monitors the signals from **Device**, looking for the bad scenario and deadlocks the system if spotted. For a programmer, this is just another function to write (though we need a bit more CSP-M):

```
Check (n) =
  if n >= 2 then STOP else
    a0 -> Check (n+1) [] d0 -> Check (0) [] d1 -> Check (0) []     -- "[]" means wait for one or more
    a1 -> Check (n)   [] b0 -> Check (n) [] b1 -> Check (n) []     -- of the operand processes to be
    c0 -> Check (n)   [] c1 -> Check (n)                           -- able to run; choose one and run.

CheckDevice = Device [| {a0, a1, b0, b1, c0, c1, d0, d1} |] Check (0)
```

The parameter to **Check** records how many **a0** signals have happened since the last **d0** or **d1**. If this reaches 2, **Check** stops and leaves **CheckDevice** deadlocked (since **Check** and **Device** must synchronise on all signals from **Device**). FDR2 quickly confirms that **CheckDevice** is free from deadlock. *Hence, the feared in-service problem cannot happen.* **Q.E.D.**

Protocol checking monitors, such as **Check**, are sometimes used live (e.g. in device drivers) to ensure correctness at run-time. It is important to note that we are using **Check** purely for static analysis – it has no role at run-time and, therefore, no impact on performance.

So far, our checks have concerned *safety* – namely that our system will not do incorrect things. More strongly, we may need to consider *liveness* – namely that our system *will do* the right thing in all circumstances. To do this, we need to check that our system *failure refines* a specification of all those right things. A CSP *failure* is a state that a system reaches where it *may* refuse to synchronise with its environment on some specified set of events. Any failure of a *failures-refinement* of a specification will be an allowed failure of that specification – therefore, OK. Here is an explicit specification of all the signal patterns **Device** must be able to perform – it's written from our intuitive understanding for **Device**'s behaviour (described in the middle of page 2):

```
DeviceSpec =
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP); (c0 -> SKIP ||| c1 -> SKIP ||| d0 -> SKIP);
  a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP); (c0 -> SKIP ||| c1 -> SKIP ||| d1 -> SKIP); DeviceSpec
```

The *interleave* operator, |||, is shorthand for the *parallel* operator with an empty ***sync-set***. It means the event sequences of its operand processes may interleave freely. Again, FDR2 quickly confirms that **Device** *failure refines* **DeviceSpec**. In fact, the reverse is also confirmed – so **Device** and **DeviceSpec** have exactly the same traces and failures. **Device** was structured as a network of three sub-processes to reflect computations (abstracted away here) naturally located within those components. **DeviceSpec** shows us *all* patterns of synchronisation **Device** *can and will* perform, on demand from its environment.

Reflection

Similar exercises to those presented here were worked though *live* in a graduate class at UNLV in the last academic year, as part of a specialist concurrency module. Previously, they had studied a range of approaches to concurrency, including material from the undergraduate *Concurrency Design and Practice* [6] course (to over 60 second year students last year at the University of Kent) discussed at last year's workshop [7]. This material on verification will be added to the Kent course next year.

By the time of this exercise, students were comfortable with using occam-π in several non-trivial projects (thousands of interacting processes). So, the example system here would be considered fairly trivial. Nevertheless, if the application were safety critical, it was appreciated that relying just on our intuition (based on understanding the low-level concurrency semantics of occam-π) was unsafe.

During the exercise, students were given an overview (through examples) of the syntax of CSP-M, with the semantics of its operators defined by relating them to occam-π syntax and semantics. The functional nature of CSP-M, as opposed to the imperative nature of occam-π, was no particular obstacle. Using FDR2 through its GUI is a bit old-fashioned (by modern GUI standards), but easy enough to manage the operations described above. The students tried their own test sequences of signals from **`Device`** and correctly obtained confirmation or rejection. Writing specific checking processes for long-term dangers (like **`Check`**) was harder, but they warmed to this with more practice with CSP-M. *What-ifs (e.g. does* **`Check`** *still hold if the* **`ping`***s are removed? Answer: no!)* could be explored without running any code. Writing suitable specifications and working on failures refinement was beyond the scope of this exercise. We hope for more time next year.

occam-π enables concurrency to be used to simplify complex system design. Its run-time system imposes memory overheads of no more than 32 bytes per process and run-time overheads for synchronisation of the order of tens of nanoseconds – and it eats multicore nodes for breakfast. Small memory/power platforms and large scale complex system modelling (millions of processes) are addressed. It teams well with CSP to provide rich and flexible analysis. Future effort will be made to tie occam-π directly with the FDR model checker, so that only one syntactic representation is needed.

An intriguing observation from this story is that real *verification* of the behaviour of communicating processes is achieved, even though we have engaged in only simple reasoning ourselves. The status of the judgements from FDR2 is formal proof (although purists may descend to complaints about the lack of formal proof of the correctness of the FDR2 implementation, the C++ compiler with which it was compiled, the computer hardware on which it was run, etc.). Those complaints apart, this verification is a significant step forward in gaining confidence in our concurrent systems – yet all we feel we have done is program! This brings concurrency verification into the realm of students and mortals. Further reading may be found in [8].

References

[1] Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall.

[2] Roscoe, A. 1997. *The Theory and Practice of Concurrency*. Prentice Hall.

[3] Welch, P. and Barnes, F. 2005. *Communicating Mobile Processes: introducing occam-π*. In "25 Years of CSP", A. Abdallah, C. Jones, and J. Sanders, Eds. Lecture Notes in Computer Science, vol. 3525. Springer, 175–210.

[4] Barnes, F.R. and Ritson, C.G. 2010. *Checking Process-oriented Operating System Behaviour using CSP and Refinement*. ACM SIGOPS Oper. Syst. Rev. 43, 4 (Jan. 2010), 45-49. www.cs.kent.ac.uk/pubs/2009/2983/

[5] Formal Systems (Europe) Ltd. 2007. *FDR2 (Failures, Divergences, Refinement) model checker,* download page: www.fsel.com/software.html

[6] Welch, P.H. 2010. *Concurrency Design and Practice*. www.cs.kent.ac.uk/projects/ofa/sei-cmu/

[7] Welch, P.H. and Barnes, F.R.M. 2009. *Concurrency First*. Position paper, *First Workshop on Curricula in Concurrency and Parallelism*, OOPSLA 2009. www.cs.kent.ac.uk/projects/ofa/cfc/phw-frmb-position.pdf (slides: www.cs.kent.ac.uk/projects/ofa/cfc/concurrency-first-cfc-2009.ppt, www.cs.kent.ac.uk/projects/ofa/cfc/concurrency-first-cfc-2009.pdf)

[8] Welch, P.H. and Pedersen, J.B. 2010. *Santa Claus: Formal Analysis of a Process-oriented Solution*. ACM Trans. Program. Lang. Syst. 32, 4, Article 14 (April 2010). http://doi.acm.org/10.1145/1734206.1734211